

國立清華大學資訊工程系 110 學年度上學期專題報告

專題名稱	Verilog Simulation Optimization via Instruction Reduction				
參加競賽 或計畫	<input checked="" type="checkbox"/> 參加對外競賽		<input type="checkbox"/> 參與其他計畫		<input type="checkbox"/> 無參加對外競賽或任何計畫
學號	107062134	107000118	107060024		
姓名	樊明勝	劉家豪	許鎧博		

摘要

Since the complexity of a SOC increases exponentially, functional verification becomes a time-consuming problem in the design cycle. To reduce the main reason contributing to the low simulation efficiency on CPU time, contestants are required to decrease the number of continuous assignments and bit-selects.

This objective is to facilitate the simulator by a parser, which performs Verilog-to-Verilog transformation while minimizing instructions. The optimized Verilog file must be syntactically correct, inviolate the given limitations, and be the functional equivalent to the input Verilog file.

Under the constraints and regulations, this project provides a different data structure to save the parsing instructions rather than traditional graphs. Our data structure having the good property to represent the circuit directly without any transformation is constructed to implement the logic optimization. This reduces the complexity during the optimization stage, contributing to exploring potential optimization strategies on the following optimization. In addition, to implement traditional optimization methods, this project provides novel strategies to further improve simulation efficiency.

As the result, one of these new approaches can get a score ten times better than before, the instructions in one testcase can be reduced to one over seventy-five compared to the original one. Even some testcases can be simplified to the simplest form of them by our work. All of the members in this project participate in the CAD contest, but each of us forms a single-member team. **Two of the members won the ICCAD awards, one is the first prize and the other is fourth prize.**

中華民國 110 年 11 月

Table of contents

1. Introduction

2. Objective & Attribution

3. Problem Formulation

3.1 Program Requirement

3.2 Original Verilog Design

3.3 Output Optimized Verilog Design

3.4 Verification of Simulation Correctness

3.5 Evaluation

4. Methodology Compared to Current Research

5. Implementation

5.1 Data Structure

5.2 Basic Logic Synthesis And Basic Logic Rules

5.3 Substitution

5.4 Pattern Comparison

5.5 Commutative Law

5.6 Further Pattern Comparison

5.7 SAT Solver Based Redundant Wire Checking

5.8 The usage of xformtmp

5.9 Gate Merging

5.10 Output Merging

6. Result

7. Conclusion

8. Reference

1. Introduction

Since the size and complexity of a SOC keep growing exponentially to satisfy various applications, functional verification gradually becomes the most time-consuming part of the whole design cycle [1]. How to perform functional verification in a stringent time-to-market window is undoubtedly a critical problem in the EDA field.

The simulator is an essential tool for functional verification from RTL to gate-level, whose performance is directly related to the IC development progress. However, when the design size exceeds a certain amount, the simulation time becomes unacceptable. IC designers may spend several weeks in gate-level simulation with millions of gates in RTL [2]. In other words, the simulator performance enhancement is beneficial for the whole industry.

The objective of this contest topic, Verilog Simulation Optimization via Instruction Reduction [3], is to enhance the performance of the simulator from the algorithm point of view. The contestants are asked to write a program performing Verilog-to-Verilog transformation to minimize instructions/assignments for performance consideration. The generated Verilog file must be syntactically correct, inviolate the given limitations, and be functional equivalent to the input file, then simulation efficiency will be evaluated..

2. Objective & Attribution

This project focuses on facilitating Verilog simulation by decreasing the number of instructions. Without a doubt, keeping the correctness of output is a necessity. Based on the efficiency of Verilog simulation is proportional to the CPU time spent on instructions execution. Simply put, if the instructions are reduced, the simulation efficiency grows.

In this problem, contestants are asked to write a C++ parser to convert a given Verilog design into an optimized Verilog design. This source-to-source transformation only needs to be considered in continuous assignments, bit/part-selects of vectors, bit operations, and wire type signals to further optimize for the given Verilog design. Consequently, the objective is to minimize the total number of continuous assignments and vector bit/part-selects in the given Verilog design.

3. Problem Formulation

As mentioned above, the contestants are required to develop a C++ program to optimize the input Verilog file. The requested program “verilogopt” takes a Verilog design “original.v” as an input and outputs the optimized Verilog design “optimized.v”. Besides, several restrictions need to be obeyed during the optimization. Only when passing the verification of simulation correctness, the program can be evaluated by the

testcases with 2-state input values, 0 & 1. The following requirement is referenced from “Verilog Simulation Optimization via Instruction Reduction”.

3.1. Program Requirement

Implementation of a Verilog parser is necessary to this problem, any third-party open-source are not allowed in the program. The input Verilog file “original.v” uses very restrictive Verilog syntax to confirm the consistency when parsing. However, the output Verilog file “optimized.v” provides flexible syntax to create innovation optimization methods.

3.2. Original Verilog Design

The input of this problem is a given Verilog file named “original.v”, which contains two modules: dut and tb. The module dut is the main module to be optimized and the module tb is merely an auxiliary module for checking the simulation correctness. There are only two packed ports, out and in, which are by default 2-state wires, 0 or 1 as follow:

```
module dut (out, in);  
    output[SIZEOUT:0] out;  
    input[SIZEIN:0] in;
```

In addition to the ports, a bunch of 1-bit wire signals is declared locally in the module dut as temporary variables as the below example.

```
wire origtmp1;  
wire origtmp2;  
...  
wire origtmpN;
```

The following expressions of ports and local wires are all continuous assignments, to avoid confusion, Backus-Naur Form (BNF) is adopted.

```
assign <LHS> = <RHS>;
```

<LHS> is either a bit-select on port-out or a temporary wire: out[#] or origtmp# (where # denotes constant).

```
<LHS> ::=  
    out[#]  
    | origtmp#
```

<RHS> is either one item, one item with the unary operation, or a binary operation with two items.

```
<RHS> ::=  
    <ITEM>  
    | <UNARY_OP> <ITEM>  
    | <ITEM> <BINARY_OP> <ITEM>
```

<ITEM> can be either a 4-state constant, a bit-select on port out, a bit-select on a port in, or a temporary wire.

```
<ITEM> ::=  
    <1-bit 4-state constant>  
    | out[#]  
    | in[#]  
    | origtmp#
```

<UNARY_OP> is the bitwise negation operator.

```
<UNARY_OP> ::=  
    ~ (bit negation)
```

<BINARY_OP> is one of the following binary bitwise operators.

```
<BINARY_OP> ::=  
    & (bit and)  
    || (bit or)  
    | ^ (bit xor)
```

Consequently, after considering all regulations above, here is an example of input Verilog file.

```
module dut (out, in);  
    output[3:0] out;  
    input[15:0] in;  
    wire origtmp1;  
    assign origtmp1 = 1'b0;  
    assign out[2] = in[0] | origtmp1;  
    assign out[3] = origtmp1 | 1'b0;  
endmodule
```

```

module tb();
    reg[3:0] results;are
    reg[15:0] feed[1];
    dut duttest(results, feed[0]);
    initial begin
        $readmemb("data.txt", feed);
        $display(results);
    end
endmodule

```

3.3. Output – Optimized Verilog Design

The output Verilog file “optimized.v” also contains two modules: dut and tb. The module dut represents the optimized instructions reduced by contestants, the module tb needs to be identical to that in “original.v”. To find innovation optimization strategies, a little flexible syntax is allowed in this problem. The followings are the regulations to syntax:

The declaration of extra temporary wires is permitted for a single or multiple-bit wide. However, multiple bits are required to be one-dimensional and packed with either ascending or descending indices.

```

// 1-bit → OK
wire xformtmp1;
// packed 4-bit, descending indices from 3 to 0 → OK
wire[3:0] xformtmp2;
// packed 7-bit, ascending indices from 0 to 6 → OK
wire[0:6] xformtmp3;
// unpacked 2-bit → NOT allowed
wire xformtmp4[1:0];
// packed 2 or more dimensions → NOT allowed
wire[7:0][1:0] xformtmp5;

```

<LHS> allows 2 more types: xformtmp# and out[MSB:LSB]. In Verilog syntax, myvect[MSB:LSB] denotes ‘part select’ of vector myvect, which means consecutive bits from index MSB to index LSB (width equals to MSB-LSB+1). According to IEEE standard, though ascending or descending indices are both fine, every use must be consistent with its declaration.

```

<LHS> ::=
    | out[#]
    | out[MSB:LSB]
    | origtmp#
    | xformtmp#
    | xformtmp#[#]
    | xformtmp#[MSB:LSB]

```

<RHS> is the same as input Verilog.

<ITEM> allows four extra types: out[MSB:LSB], in[MSB:LSB], xformtmp#, and xformtmp#[MSB:LSB]. Additionally, it could be a N-bit 2-state constant.

```

<ITEM> ::= <1 or N-bit 2-state constant>
    | out[#]
    | in[#]
    | origtmp#
    | out[MSB:LSB]
    | in[MSB:LSB]
    | xformtmp#
    | xformtmp#[MSB:LSB]

```

<UNARY_OP> is the same as input Verilog.

<BINARY_OP> is the same as input Verilog.

3.4. Verification of Simulation Correctness

A given input file “data.txt” which contains a one-line text string of characters: 0, 1 is used to verify the correctness of simulation. For instance, 0111100100011001 may the data to read in is for ‘reg[15:0] data[1]’. Simulation results, i.e., the value of out-port vectors are dumped via \$writememb. Simulation correctness is checked by comparing the \$writememb results of “original.v” and “optimized.v”. In this project, we use ABC: A System for Sequential Synthesis and verification as the formal verification tool when facing the small circuit. For big circuit, we only can use the Synopsys VCS tool stimulate our circuit and compare the result.

Suppose we have the following “original.v”.

```

module dut (out, in);
    output[3:0] out;
    input[15:0] in;

```

```

    wire origtmp1;
    assign origtmp1 = 1'b0;
    assign out[2] = in[0] | origtmp1;
    assign out[3] = in[1] | 1'b0;
endmodule

```

- The “original.v” contains 3 continuous assignments of origtmp1, out[2], out[3] and 5 bit-selects of 1'b0, in[0], origtmp1, in[1], 1'b0.

Then we may derive the following 3 versions of valid “optimized.v”, which represents the different degrees of the optimization.

Version 1:

```

    module dut (out, in);
        output[3:0] out;
        input[15:0] in;
        assign out[2] = in[0] | 1'b0;
        assign out[3] = in[1] | 1'b0;
    endmodule

```

- The “optimized.v” contains 2 continuous assignments and 4 bit/part selects: out[2], in[0], out[3], in[1].

Version 2:

```

    module dut (out, in);
        output[3:0] out;
        input[15:0] in;
        wire[1:0] xformtmp1;
        assign xformtmp1 = 2'b0;
        assign out[3:2] = in[1:0] | xformtmp1;
    endmodule

```

- The “optimized.v” contains 2 continuous assignments and 2 bit/part selects: out[3:2], in[1:0].

Version 3:

```

    module dut (out, in);
        output[3:0] out;
        input[15:0] in;
        assign out[3:2] = in[1:0];
    endmodule

```


- The “optimized.v” contains 1 continuous assignments and 4 bit/part selects: out[3:2], in[1:0].

3.5. Evaluation

For each “optimized.v”, a final score will be concluded from simulation correctness, optimization regulation, execution time limit, and simulation efficiency. Simulation correctness, execution time limit, and optimization regulation are priority requirements. If any of them is violated, simulation efficiency will NOT be evaluated any further due to the content of that matrix being set to all zero.

Simulation correctness is evaluated by CSCORE, if any differences are found to the “original.v”, then CSCORE will be 0, whereas CSCORE is set to 1.

- CSCORE = 1 (if no diffs)
- CSCORE = 0 (if any diffs are found)

Optimization regulation is represented by RSCORE, if any regulation is violated, then Rscore will be 0, vice versa.

- RSCORE = 1 (if all regulations are followed)
- RSCORE = 0 (if any regulation is violated)

Execution time limit is set as 600 seconds that cannot be exceeded for each testcase.

- TSCORE = 1 (if the program executes 600sec or less)
- TSCORE = 0 (if the program executes longer than 600sec)

Simulation efficiency is measured with the total number of two types of assignments in the module dut in “optimized.v” as follows:

- continuous assignments (denoted as <COUNT_ASGN>)
- bit/part-selects appearing in all continuous assignments (denoted as <COUNT_SELS>)
- $ESCORE = (\langle OUT_PORT_WIDTH \rangle + \langle IN_PORT_WIDTH \rangle) / (\langle COUNT_ASGN \rangle + \langle COUNT_SELS \rangle)$

The number of ($\langle OUT_PORT_WIDTH \rangle + \langle IN_PORT_WIDTH \rangle$) is fixed in each “original.v”. The lower the number of ($\langle COUNT_ASGN \rangle + \langle COUNT_SELS \rangle$) indicating fewer CPU instructions needed, the higher the ESCORE is. And FSCORE represents the final score of each testcase like the following.

$$FSCORE = CSCORE * RSCORE * TSCORE * ESCORE$$

To show how various optimization methods influence testcases with different scores, there are three versions of “optimized.v” in the following:

Version 1:

```
module dut (out, in);
    output[3:0] out;
    input[15:0] in;
    assign out[2] = in[0] | 1'b0;
    assign out[3] = in[1] | 1'b0;
endmodule
```

- The “optimized.v” contains 2 continuous assignments and 4 bit/part selects: out[2], in[0], out[3], in[1]. $\langle \text{COUNT_ASGN} \rangle + \langle \text{COUNT_SELS} \rangle = 2 + 4 = 6$.

Version 2:

```
module dut (out, in);
    output[3:0] out;
    input[15:0] in;
    wire[1:0] xformtmp1;
    assign xformtmp1 = 2'b0;
    assign out[3:2] = in[1:0] | xformtmp1;
endmodule
```

- This “optimized.v” contains 2 continuous assignments and 2 bit/part selects: out[3:2], in[1:0]. $\langle \text{COUNT_ASGN} \rangle + \langle \text{COUNT_SELS} \rangle = 2 + 2 = 4$.

Version 3:

```
module dut (out, in);
    output[3:0] out;
    input[15:0] in;
    assign out[3:2] = in[1:0];
endmodule
```

- The “optimized.v” contains 1 continuous assignments and 2 bit/part selects: out[3:2], in[1:0]. $\langle \text{COUNT_ASGN} \rangle + \langle \text{COUNT_SELS} \rangle = 1 + 2 = 3$.

As the number of ($\langle \text{OUT_PORT_WIDTH} \rangle + \langle \text{IN_PORT_WIDTH} \rangle$) is fixed, the version 3 of all “optimized.v” leads to the smallest ($\langle \text{COUNT_ASGN} \rangle + \langle \text{COUNT_SELS} \rangle$) and highest FSCORE, 6.67 among all three versions of “optimized.v”

Table 1. Different FSCORE to each “optimized.v”

Version	Version 1	Version 2	Version 3
<COUNT_ASGN> + <COUNT_SLES>	6	4	3
FSCORE	$20/6 = 3.33$	$20/4 = 5$	$20/3 = 6.67$

Both public and hidden testcases will be used to evaluate a contestant’s program. The ranking of the contest is based on the summation of the normalized FSCORE compared to all contestants from each testcase. For instance, if the highest FSCORE in testcase 1 is 0.1 and your FSCORE is 0.01, then the highest FSCORE will be set to 100, and your normalized FSCORE will be 10.

4. Methodology Compared to Current Research

In recent years, researchers have usually used traditional graphs such as AIG to do logic optimization [4]. However, we use a data structure similar to the binary tree to represent the circuit rather than AIG. The complexity of traversing the circuit will be decreased by this data structure, which has good properties to deal with don’t care and pattern comparison implementation

What’s more, we find that there are few kinds of research talking about how to optimize the RTL file which has the XOR gates. In our program, we develop this structure to optimize the Verilog file and the circuit accompanied by the XOR gates. Fig. 1 is the flowchart of our work, which will be elaborated on in the next part.

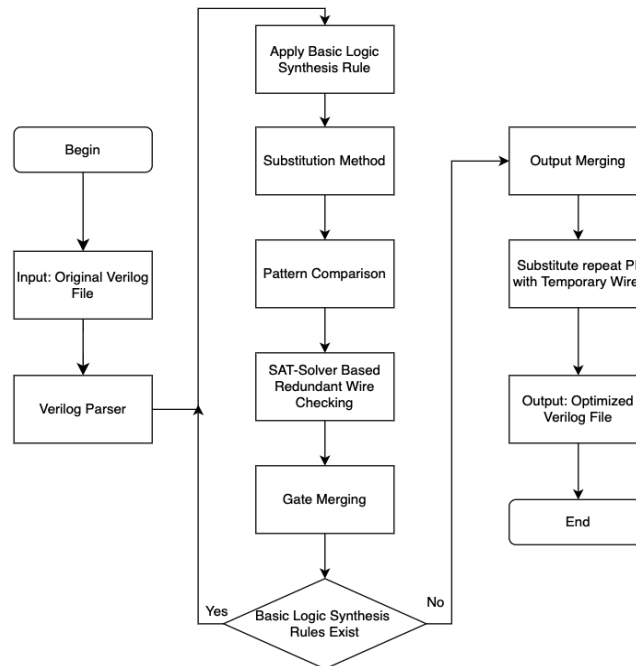


Figure. 2 Flowchart

5. Implementation

The implementation procedure is followed the flowchart above. First of all, our parser will take the given Verilog file as input, then build a data structure similar to the binary tree. Basic logic rules and the substitution method are applied to obtain a reduced Verilog file and start to implement local optimization.

We first consider to minimize on the tree structure, some Boolean identities are used to find the subtree patterns of the functional equivalent equations, then the structure of the subtree can be rewritten to optimize the circuit to implement pattern matching. After replacing the tree structure, SAT-solver-based redundant wire checking focuses on removing the wires.

Gates merging is to compare two circuits and merge gates that have the same functionality. Output merging is an operation to combine the continuous output with the same type in both input and output. Last but not least, we substitute repeated prime implicant with temporary wires, Xformtmp, then output the optimized Verilog file.

5.1. Data Structure

We define a node with the following information: operation, left type, left value, right type, right value, and flag as Figure. 2. A node is used to record one assignment of the input Verilog file. Before using the parser to extract the information, we will read the information of the input file such as the input and output size, and the number of the origtmp size. After we have read the input information, we will construct two arrays (output array and origtmp array). The data structure of a cell of the array will be the node we have constructed previously.

The right_type and left_type of the node structure will be three conditions (origtmp, in, constant). The right_value and left_value will denote the index if the type is origtmp or in; those will be the constant value (1'b1, 1'b0, 1'bz, 1'bx) if the type is const (constant).

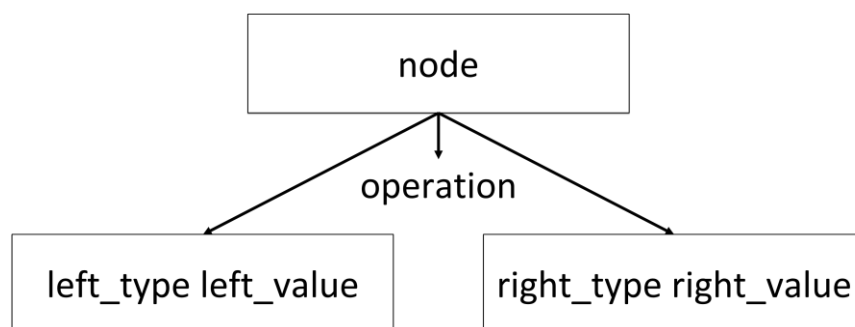


Figure. 2 Data Structure

Once we finish the parser of the input file and get all information of the input assignments, we will now consider constructing the type of data structure. Since we

would have the perfect structure that would only contain two inputs in one assignment. Therefore, for each output node (out[#]), we can view it as the root of the binary tree.

Before we construct the tree, we have to define the tree node of the binary tree. With the data structure of the binary tree, we will next construct a binary for each output node with the help of the previous constructed arrays (output and origtmp). We use the “factoringTree” function to implement.

Be based on the left type/value and right type/value to point the current node’s pointer to the right array cell. Therefore, we will get the final binary tree. If the size of the output is M and the size of origtmp is N, the complexity of the factoring tree will be $O(M+N)$ since we will read the arrays to factor the tree. Also, since we will go through the entire tree to traverse the node, the complexity of traversal will be $O(M+N)$.

5.2. Basic Logic Synthesis And Basic Logic Rules

In Boolean algebra, we all know several Boolean identities are used to minimize the Boolean function. First of all, we will list some common constant operations of the Boolean function in the following table.

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Figure. 3 Basic logic operations

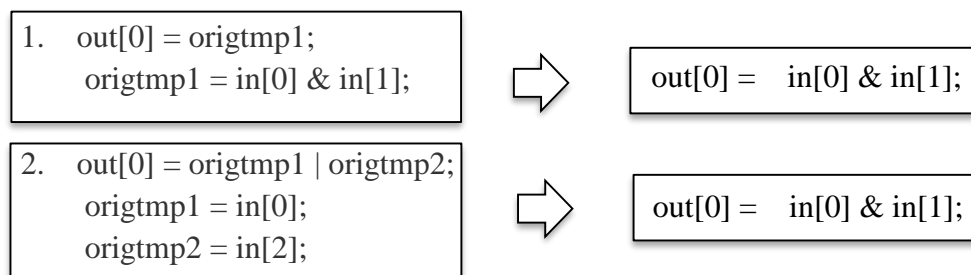
Therefore, we will go through all the cells of the array to see whether there is possible bits operation in the first step minimization. Also, note that high z signals and don’t care signals will have the same behavior in the table. Therefore, we will view those two signals in the same variable (constant 2). When a cell of the array consists of a constant and a variable, the Boolean identities also tell us that we can use those identities to minimize the function as follow:

1. $A|0 = A$
2. $A|1 = 1$

3. $A|2 = \text{either } 1 \text{ or } 2 \text{ (} = 1 \text{ when } A \text{ is } 1\text{)}. \text{ For simplicity, } A|2 = 1$
4. $A \& 0 = 0$
5. $A \& 1 = A$
6. $A \& 2 = 0$ (same reason as 3.)
7. $A^0 = A$
8. $A^1 = \sim A$
9. $A^2 = 2$ (by the previous chart)

5.3. Substitution

After minimizing the inner data structure, we hope to combine or substitute some variables with the corresponding equations. For example, if we have two equations:



It is obvious that we can substitute origtmp1 with $\text{in}[0] | \text{in}[1]$ into $\text{out}[1]$ for the first case. When we represent the equations as our data structure, we can see the relationship between the two equations. Also, we will hope that we will start our minimization from the leaves; hence, we will first use the postorder traversal to start the minimization from roots.

In the substitution method, the worst case will be going through all nodes of the binary tree. Therefore, if the size of the output is M and the size of origtmp is N , then the overall complexity of second step minimization will be $O(M+N)$.

5.4. Pattern Comparison

In the third step minimization, we offer several Boolean equations that can be minimized as the equations below. If we use tree traversal to find the subtree patterns of the equations, then we can rewrite the structure of the subtree to optimize the circuit.

1. $a|(a \& b) = a$
2. $a \& (a|b) = a$
3. $a|(\sim a \& b) = a|b$
4. $a \& (\sim a|b) = a \& b$
5. $\sim a \& b | a \& \sim b = a^b$
6. $\sim(\sim a) = a$ (Self-Negation)
7. $a \& (\sim a) = 0$ (Double Negation)

For example, the previous equations will have the structure like previously the following graph:

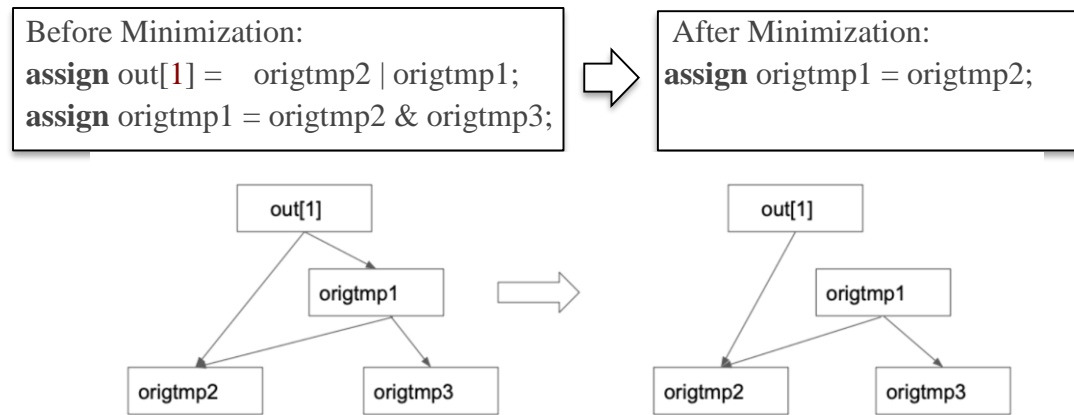


Figure. 5 The pattern comparison example of $a|(a \& b) = a$

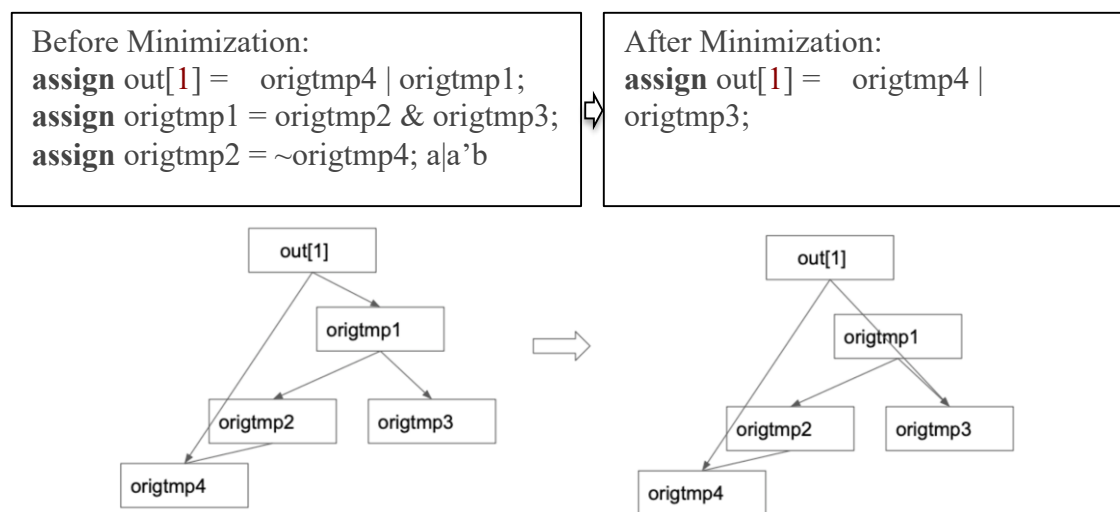
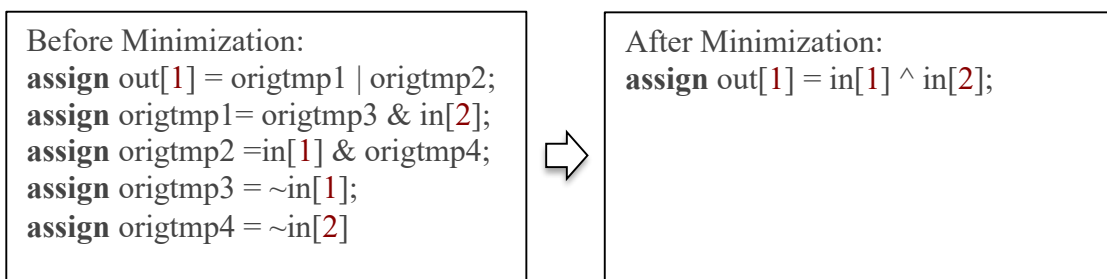


Figure. 6 The pattern comparison example of $a|(\sim a \& b) = a|b$



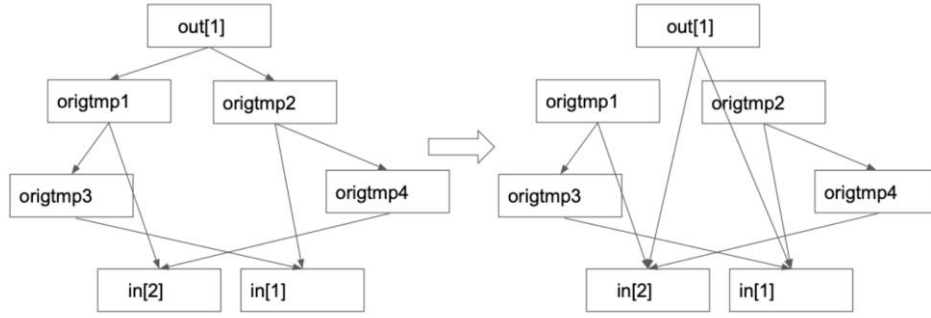
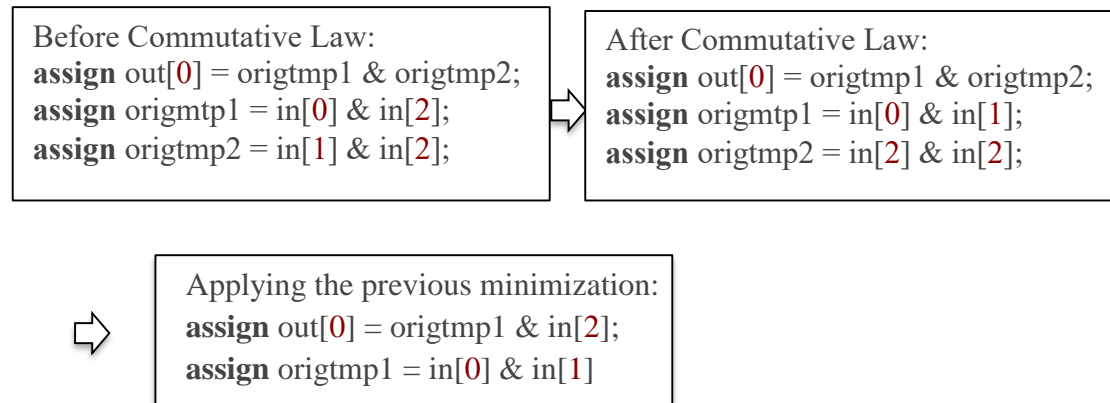


Figure. 7 The pattern comparison of $\sim a \& b \mid a \& \sim b = a \wedge b$

5.5. Commutative Law

In the Boolean equation, several gates have the commutative identity. Therefore, it is likely to get an optimized function with the help of commutative law. For example,



With the help of pattern comparison, we can get the optimized circuit in the simple tree traversal. However, if we use the method to revise the circuit, we will change the functionality of the circuit (We have changed the functionality of origtmp1 and origtmp2). Therefore, we will need to construct an additional wire to help us implement the commutative law. The equation should be revised as:

Also, we will hope that we will start our minimization from the leaves; hence, we will first use the postorder traversal to start the minimization from roots. Note that we can only use origtmp, xformtmp wire in the output file. Therefore, the additional wires should only be named as xformtmp. The dynamic method to allocate the xformtmp can be referred to the following graph:

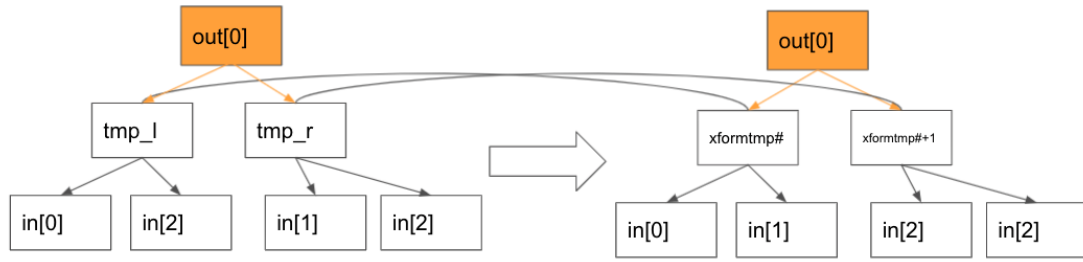


Figure. 8 Commutative Law

Also, we will hope that we will start our minimization from the leaves; hence, we will first use the postorder traversal to start the minimization from roots. In the third step minimization, the worst case will be going through all nodes of the binary tree. Therefore, if the size of the output is M and the size of origtmp is N , then the overall complexity of second step minimization will be $O(M+N)$.

5.6. Further Pattern Comparison

In further pattern comparison, we offer several Boolean equations that can be minimized as the equations below. If we use tree traversal to find the subtree patterns of the equations, then we can rewrite the structure of the subtree to optimize the circuit.

1. $a \wedge (a \& b) = a \& \sim b$;
2. $a \wedge (a | b) = \sim a \& b$;
3. $a \& (a \wedge b) = a \& \sim b$;
4. $a | (a \wedge b) = a | b$;
5. $a \wedge \sim (a \& b) = \sim a | b$;
6. $a \wedge \sim (a | b) = a | \sim b$;
7. $a \& (a \wedge b) = \sim a \& b$;
8. $a | (a \wedge b) = \sim a | \sim b$;

We will use pattern comparison the same as the previous pattern comparison; however, we will use the additional wires to avoid the revision of the functionality. We will hope that we will start our minimization from the leaves; hence, we will first use the postorder traversal to start the minimization from roots.

5.7. SAT Solver Based Redundant Wire Checking

In the EDA field, we will use testing (stuck-at-fault) to check the possible faults in the chips. However, stuck-at-fault can also help us to find out the redundant wire of the circuit. For a combination circuit, an undetectable fault is corresponding to a redundant wire. Undetectable faults do not change the functionality of the circuit. The reason can be referred to here.

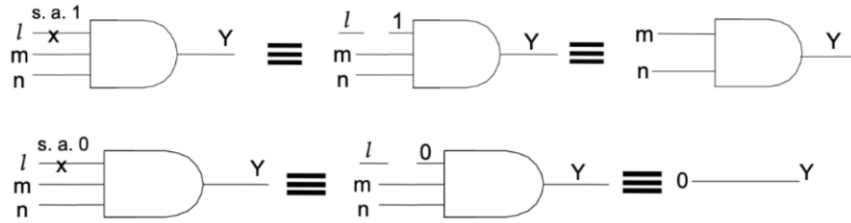


Figure. 9 Remove redundant wires by stuck-at-fault test

The previous graph shows when the s-a-1 fault is assigned to an AND gate. No matter which value m and n are, wire l can always be viewed as the redundant wire. On the other hand, we will get three redundant wires when wire l is assigned to constant 0.

How can we say the wire is redundant? As previously mentioned, we will get the same functionality as the original circuit and the stuck-at-fault circuit. Therefore, we will need to generate the whole possible inputs to compare the functionality of the original circuit and the stuck-at-fault circuit.

Note that we will use the XOR gate to check two circuits to check whether the circuits are equivalent or not to the following equations.

$$f = ab+ac, fa = ac$$

T_a = the set of all tests for fault a

$$= \text{ON_set}(f \wedge fa)$$

$$= \text{ON_set}(f) * \text{OFF_set}(fa) + \text{OFF_set}(f) * \text{ON_set}(fa)$$

$$= \{(a, b, c) \mid (ab+ac)(ac)' + (ab+ac)'(ac) = 1\}$$

$$= \{(a, b, c) \mid abc' = 1\}$$

$$= \{(110)\}$$

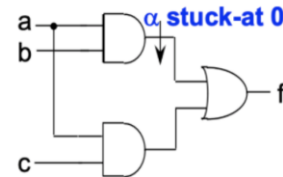


Figure. 10 Stuck-at-fault test example

In the final equation of the example, we tend to find the possible combination of a, b, and c for equation $abc' = 1$. Note that this is a SAT problem; therefore, we will use an SAT solver to check the circuit. We can find lots of open-source modern SAT solvers. With the help of modern SAT solvers, we can efficiently check if the wire is redundant or not. In this project, we will use Glucose SAT.

In order to implement the SAT-based redundant wire checking, we will need to first duplicate the original circuit for a stuck-at-fault circuit.

After we duplicate the tree, we will need an additional XOR gate to connect two circuits into a miter circuit. Then, we can next assign either s-a-1 or s-a-0 fault into an assigned

wire. When finishing assigning the wire, we will use Tseitin transformation to transform the miter circuit into a form that can be used by SAT solver.

Tseitin transformation takes as input an arbitrary combinational logic circuit and produces a Boolean formula in conjunctive normal form (CNF), which can be solved by a CNF-SAT solver. The length of the formula is linear in the size of the circuit. Input vectors that make the circuit output “true” are in 1-to-1 correspondence with assignments that satisfy the formula. This reduces the problem of circuit satisfiability on any circuit (including any formula) to the satisfiability problem on 3-CNF formulas.

The one-to-one relation of the logic gate and Tseitin transformation is shown in the following graph.







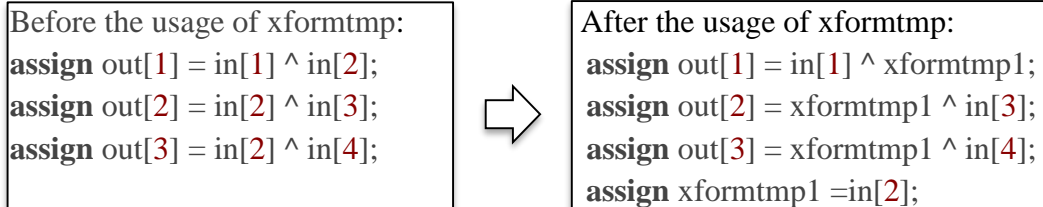
Type	Operation	CNF Sub-expression
 AND	$C = A \cdot B$	$(\bar{A} \vee \bar{B} \vee C) \wedge (A \vee \bar{C}) \wedge (B \vee \bar{C})$
 NAND	$C = \overline{A \cdot B}$	$(\bar{A} \vee \bar{B} \vee \bar{C}) \wedge (A \vee C) \wedge (B \vee C)$
 OR	$C = A + B$	$(A \vee B \vee \bar{C}) \wedge (\bar{A} \vee C) \wedge (\bar{B} \vee C)$
 NOR	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\bar{A} \vee \bar{C}) \wedge (\bar{B} \vee \bar{C})$
 NOT	$C = \bar{A}$	$(\bar{A} \vee \bar{C}) \wedge (A \vee C)$
 XOR	$C = A \oplus B$	$(\bar{A} \vee \bar{B} \vee \bar{C}) \wedge (A \vee B \vee \bar{C}) \wedge (A \vee \bar{B} \vee C) \wedge (\bar{A} \vee B \vee C)$
 XNOR	$C = \overline{A \oplus B}$	$(\bar{A} \vee \bar{B} \vee C) \wedge (A \vee B \vee C) \wedge (A \vee \bar{B} \vee \bar{C}) \wedge (\bar{A} \vee B \vee \bar{C})$

Figure. 11 Tseitin transformation

We now can use the Tseitin transformation to build the clause for the input of the SAT solver. We have mentioned the one-to-one relation between the original gates and the tseitin form. Therefore, we can apply the SAT solver to find the redundant wire.

5.8. The usage of xformtmp

First usage: The `in[1]` would be counted as one cost for bit selection, but `xformtmp` wouldn't. Thus, we use the `xformtmp` to substitute the input node which appears multiple times in the primary input of other gates, like `xformtmp1 = in[1]`; The second usage mentioned in the “Output merging” operation and the “`xformtmp`” is the new temporary wire of sequential node.



- The score of the original one, counting `assign` = 3, bit selection = 9
- The score of optimized one, counting `assign` = 4, bit selection = 5

From the above, we see the optimized one has the better score. The second usage mentioned in the “Output merging” operation and the “`xformtmp`” is the new temporary wire of sequential node.

5.9. Gate Merging

Gates merging is to compare two circuits and merge gates that have the same functionality for each node one by one. If the program finds two nodes that have the same child node, put the parent nodes' pointer of one of them onto another one. The process is bottom-up. Time complexity is $O(n^2)$. Because the time complexity isn't well, we need to implement an algorithm like simulated annealing and a limit for this process.

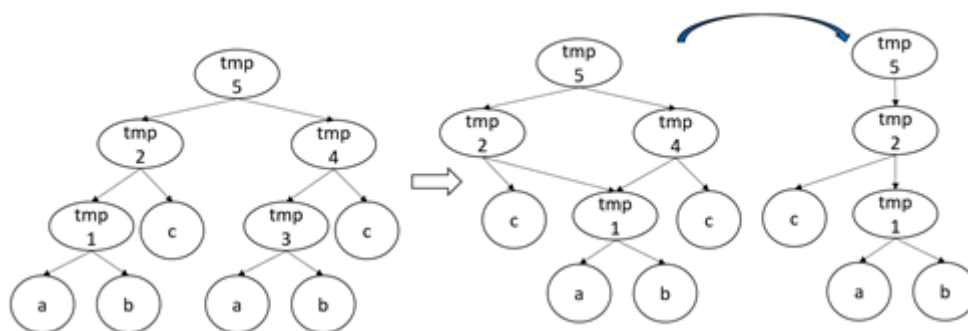


Figure. 12 Gate merging

↑ Assume the operators of `tmp1` and `tmp3` are the same and the `tmp2`, `tmp4` also.

5.10. Output Merging

Output merging is an operation to combine the continuous output with the same type in both input and output. For example, the equations $\text{out}[1] = 1'b1$, $\text{out}[0] = 1'b0$ should be combined together into $\text{out}[1:0] = 2'b10$ for better score.

According to the evaluation of problem spec, we should put the primary output in a sequence form in Verilog for a better score. The equations, $\text{out}[1] = 1'b1$, $\text{out}[0] = 1'b0$, should be combined together into $\text{out}[1:0] = 2'b10$.

What's more, the problem is more complicated, $\text{out}[1] = a \wedge b$, $\text{out}[2] = c \wedge d$, $a = \text{in}[1] \wedge \text{in}[5]$, $b = \text{in}[3] \wedge \text{in}[7]$, $c = \text{in}[2] \wedge \text{in}[6]$, $d = \text{in}[4] \wedge \text{in}[8]$, should be combine as $\text{out}[2:1] = \text{xformtmp1}[1:0] \wedge \text{xformtmp2}[1:0]$, $\text{xformtmp1}[1:0] = \text{in}[2:1] \wedge \text{in}[6:5]$, $\text{xformtmp2}[4:3] = \text{in}[4:3] \wedge \text{in}[8:7]$.

The first step is to check the primary output node if it is a sequence. The second step is putting “a” and “c” wires together and “b”, “d” wires are also. The reason why we can put a and c wire together is that the child nodes are an input sequence and they have the same operators. When once find the above relation, we put them together and create a new temporary wire, and this new wire I call it “sequential node.” Then, if the child node of “a” wire isn't an input node, we should take its child node as a sequential node at beginning until the program processes the children and get the real answer back. Thus, this whole process is an iteration from the primary output to input and get the result from the bottom node.

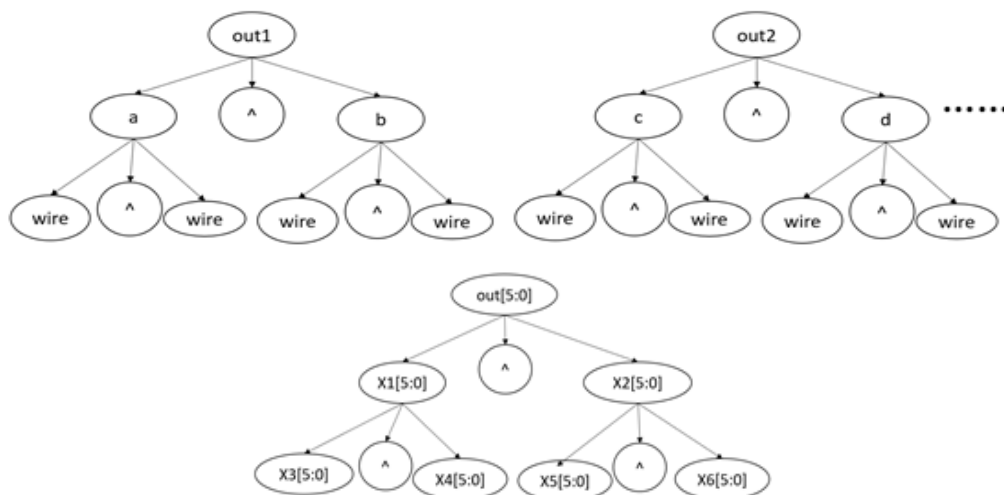


Figure. 13 Output merging with sequential X nodes

Also, we found the scoring of the output sequence has a connection with their primary input nodes. If one node in the process isn't sequential, the total structure cannot form a perfect sequence and it would increase the cost of counting bit selections but sometimes still can decrease the total score. Below have a few examples:

Before Output Merging:

```
out[1] = a ^ b;  
out[2] = c ^ d;  
a = in[2] ^ in[4];  
b = in[11] ^ in[13];  
c = in[1] ^ in[8];  
d = in[12] ^ in[14];
```



After Output Merging:

```
out[2:1]=xformtmp1[1:0] ^ xformtmp2[1:0];  
xformtmp1[0]=in[2] ^ in[4];  
xformtmp1[1]=in[1] ^ in[8];  
xformtmp2[1:0] = in[12:11] ^ in[13:14];
```

- The score of original one, counting assign = 6, bit selection =10
- The score of optimized one, counting assign=4, bit selection= 12

From the above, we can know that if the sequence isn't the perfect sequence. The optimized one has a higher bit selection cost but a lower assign cost. Then have on to the next example.

Before Output Merging:

```
out[1] = a ^ b;  
out[2] = c ^ d;  
out[3] = e ^ f;  
a = in[2] ^ in[5];  
b = in[3] ^ in[7];  
c = in[11] ^ in[14];  
d = in[12] ^ in[15];  
e = in[4] ^ in[9];  
f = in[13] ^ in[16]
```



After Output Merging:

```
out[3:1] = xformtmp1[2:0] ^ xformtmp2[2:0];  
xformtmp1[0] = in[2] ^ in[5];  
xformtmp1[1] = in[3] ^ in[7];  
xformtmp1[2] = in[4] ^ in[9];  
xformtmp2[2:0] = in[13:11] ^ in[16:14];
```

- The score of original one, counting assign = 9, bit selection =15
- The score of optimized one, counting assign=5, bit selection= 15

This example shows that the optimized one has a better score even though it has the higher bit selection cost. This one also shows that it has a longer sequence than the first example and both of their output nodes have one side that isn't the sequential node but the longer sequence one has the better performance.

Before Output Merging:

```
out[1] = origtmp1 ^ origtmp2;  
out[2] = origtmp3 ^ origtmp4;  
origtmp1 = e ^ f;  
origtmp3 = in[3] ^ in[7];  
origtmp2 = in[11] ^ in[14];  
origtmp4 = in[12] ^ in[15];  
e = in[4] ^ in[9];  
f = in[13] ^ in[16];
```



After Output Merging:

```
out[2:1] = xformtmp1[1:0] ^ xformtmp2[1:0];  
xformtmp1[0] = e ^ f;  
xformtmp1[1] = in[3] ^ in[7];  
xformtmp2[1:0] = in[12:11] ^ in[15:14];  
e = in[4] ^ in[9];  
f = in[13] ^ in[16];
```

- The score of original one, counting assign = 8, bit selection = 12
- The score of optimized one, counting assign=6, bit selection= 14

This one is actually like the first example. The original one and the optimized one have the same “e” and “f” wires. Using the in[2], in[4] to replace the e, f wires, the circuit would be the first example.

Before Output Merging:

```
out[1] = a ^ b;  
out[2] = c ^ d;  
a = e ^ f;  
c = g ^ h;  
b = in[11] ^ in[14];  
d = in[12] ^ in[15];  
e = in[4] ^ in[9];  
f = in[0] ^ in[6];  
g = in[1] ^ in[5];  
h = in[3] ^ in[7];
```



After the first transition:

```
out[2:1] = xformtmp1[1:0] ^ xformtmp2[1:0];  
xformtmp1[1:0] = xformtmp3[1:0] ^ xformtmp4[1:0];  
xformtmp2[1:0] = in[12:11] ^ in[15:14];  
xformtmp3[1] = in[4] ^ in[9]; -> not sequential  
xformtmp3[0] = in[1] ^ in[5];  
xformtmp4[1] = in[3] ^ in[7]; -> not sequential  
xformtmp4[0] = in[0] ^ in[6];
```



After the second transition:

```
out[2:1] = xformtmp1[1:0] ^ xformtmp2[1:0];  
xformtmp1[1] = origtmp7 ^ origtmp8;  
xformtmp1[0] = origtmp5 ^ origtmp6;  
xformtmp2[1:0] = in[12:11] ^ in[15:14];  
origtmp5 = in[4] ^ in[9];  
origtmp6 = in[0] ^ in[6];  
origtmp7 = in[1] ^ in[5];  
origtmp8 = in[3] ^ in[7];
```

The circuit through the first transition, which takes the wire as sequential first, would have a worse performance but the circuit through the second transition would have the meaning of the fourth example. Thus, the circuit through the second transition and the original one are the same cost.

From the above, I define the node how to return its property when the program processes the node. If the operator of the optimized node is “not” or no operator and the node have a sequential node, it should return “It is a sequential node” to its parent node. If the operator of the optimized node is “|”, “&” or “^” and the node has two sequential nodes, it should return “It is a sequential node”. The else conditions would make the node return “It isn’t a sequential node”. With the iteration in the program, the output node would know whether it can form a sequence or not. In addition, I would like to evaluate the cost after the progression and compare the optimized one and the original one and make the optimized one has a better score.

6. Result

The following experimental results in the public 9 testcases were conducted in the Linux environment provided by the CAD Contest as Table 2, which represents the score comparison between the original Verilog file and the optimized one.

One of these approaches can get a score over ten times better than the original case. For the final test, we have seventeen testcases over 96 normalized FSCORE over twenty testcases, and nine testcases get 100 normalized FSCORE. Even some test cases can be simplified to the simplest form of them by our work. As the result, After the “Final Test”, **we won the ICCAD Contest First and Fourth prizes.**

Table 2. The FSCORE of each testcases

version	original	optimized	version	original	optimized
case 1	0.365854	0.566038	case 6	0.000499875	0.0376689
case 2	0.410256	0.610687	case 7	0.397772	0.642467
case 3	0.404218	0.633609	case beta 1	0.523256	1.95652
case 4	0.0714286	0.571429	case beta 2	0.106956	0.123919
case 5	0.444444	4			

7. Conclusion

Based on the result mentioned above, our project not only implements traditional optimization methods but provides novel strategies to further improve the simulation efficiency. One of these approaches can get a score almost ten times better than before, a test case can be reduced to one over seventy-five compared to the original testcase. Even some test cases can be simplified to the simplest form of them by our work.

Our data structure contributes to decreasing the complexity of traversing the circuit, which has the great property that it can directly represent the circuit without any transformation. For example, to transform the circuit as AIG, the synthesis tool will add additional inverters into the circuit since there will only be two types of the gate in the circuit (And and Inverter). Not only provides properties to deal with don't care and rewrite subtree structure, but further optimization can also further optimization can be easily implemented.

Besides, our methodology optimizes on different aspects. The substitution method replaces some variables with the corresponding equations. For the substructure minimization, Boolean identities are implemented by the pattern comparison. From the perspective of removing unnecessary wires, SAT-solver-based redundant wire checking is used. As node simplification, gate merging decreases the number of functional equivalent gates by combining them. Furthermore, Xformtmp is added to increase the flexibility of the optimization to create potential strategies.

In the prospects, this work will calculate mandatory assignments to implement the redundant addition and removal technique or the minimization methods like fast node merging [5], which completes the node simplification since we already use the stuck-at-fault test. These public testcases will be further compared to the mainstream optimization methods, then learn the different skills from them. Based on the knowledge we learned, a better algorithm will be constructed for faster simulation efficiency. Since the data structure can be viewed as a directed acyclic graph (DAG), a DAG-aware synthesis method [6] probably is a good direction to develop in. The ultimate goal is to build a tool similar to "ABC: A System for Sequential Synthesis and Verification" [7], which provides both functions like logic synthesis and formal verification.

8. Reference

- [1] A. Evans et al., "Functional verification of large ASICs," Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175), San Francisco, CA, USA, 1998, pp. 650-655, DOI: 10.1145/277044.277210.
- [2] Y. Deng, "GPU Accelerated VLSI Design Verification," 2010 10th IEEE International Conference on Computer and Information Technology, Bradford, 2010, pp. 1213-1218, DOI: 10.1109/CIT.2010.219.
- [3] K. Ko, T. Liu, "Problem F: Verilog Simulation Optimization via Instruction Reduction", Synopsys, Inc.
- [4] L.s Machado, J. Cortadella, "Boolean Decomposition for AIG Optimization", GLSVLSI '17: Proceedings of the on Great Lakes Symposium on VLSI 2017May 2017 pages 143-148, DOI: 10.1145/3060403.3060420.
- [5] Y.C. Chen and C.Y. Wang, "Fast Node Merging With Don't Cares Using Logic Implications", IEEE Transactions on Computer-Aided Design of Integrated

Circuits and Systems (Volume: 29, Issue: 11, Nov. 2010), pp. 1827-1832, DOI: 10.1109/TCAD.2010.2058510.

- [6] A. Mishchenko, S. Chatterjee et al., “DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis”, 2006 43rd ACM/IEEE Design Automation Conference, San Francisco, CA, USA, 2006, DOI: 10.1145/1146909.1147048.
- [7] A. Mishchenko, S. Chatterjee et al., “FRAIGs: A Unifying Representation for Logic Synthesis and Verification”, Department of EECS University of California, Berkeley.